Phaser-Einstieg - Jump&Run

Hier werde ich kurz den Aufbau eines Phaser-Games erklären, bzw. werden wir so nach und nach einige Elemente kennenlernen, die man in Games einsetzt.

Inhalt

1. Konfiguration / Konfigurationsobjekt

```
var config = {
    type: Phaser.AUTO, //Welche Grafikvariante
    width: 800, //Breite des sichtbaren Bereichs
    height: 600, //Höhe des sichtbaren Bereichs
    physics: { //Welche Physics-Engine wird eingesetzt
        default: 'arcade', //Arcade, Impact oder Matter
        arcade: {
            gravity: { y: 300 }, //Gravitationseinstellungen
            debug: true //Debugmodus: Bodies und Geschwindigkeitsvektoren anzeigen
        }
    },
    scene: {
        preload: preload, //Name der Methode für das preload
        create: create, //Name der Methode für das create
        update: update //Name der Methode für das update (die Loop-Methode)
     }
};
```

Ein wichtiger Teil ist die Anfangskonfiguration, die man in einer Variable (einem Objekt) speichert. Bezeichnenderweise benutzen wir da den Namen `config`.

Hier werden die Grundeinstellungen für die Game-Engine festgelegt. Näheres habe ich oben in den Kommentaren kurz beschrieben.

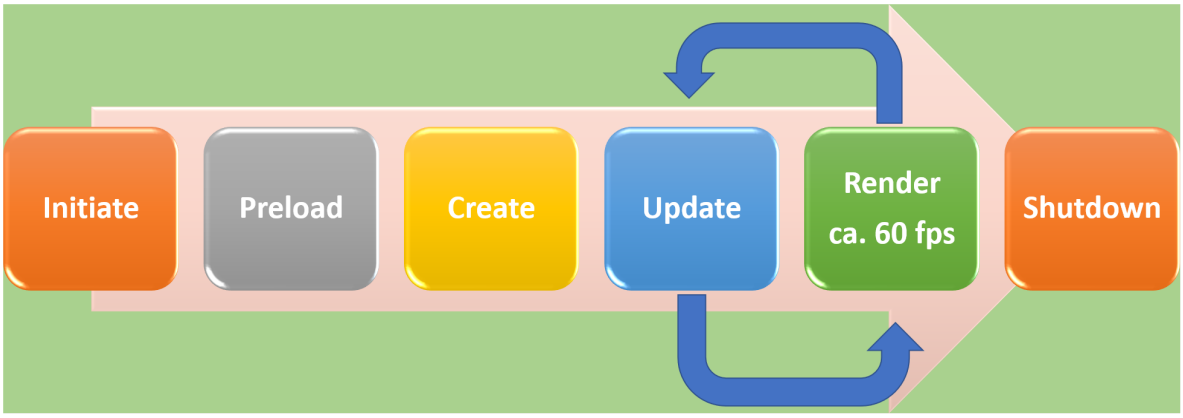
2. Globale Variablen

```
var game = new Phaser.Game(config);
var scene, player, platforms;
var cursors;
```

Für die wichtigsten Objekte im Game sollte man globale Variable definieren, die dann in allen Methoden zur Verfügung stehen. Neben dem Game-Objekt, welches hier dann auch sofort mit `new Phaser.Game(config)` initialisiert wird sind dies bei uns die Variablen für die Szene, den Spieler, die Plattformen (eine statische Gruppe) und die Tasten (Pfeiltasten für die Steuerung)

3. Der Ablauf in Phaser - Die Game-Loop

In Phaser - genauso wie in anderen Game-Engines - gibt es Prozesse (bzw. Funktionen), die in einer bestimmten Reihenfolge ausgeführt bzw. auch wiederholt werden. Hier eine Grafik dazu:



4. Preload-Methode

In der preload-Methode lädt man die Assets, die im Spiel benötigt werden. Bei uns sind das - jetzt zu Beginn - die Grafiken für den Hintergrund und die Plattformen (jeweils ein Image), sowie für die Spielfigur eine Spritesheet-Grafik - Das ist eine Grafik mit mehreren Bildern, welche unterschiedliche Zustände von Animationen darstellen. Game-Engines wie Phaser teilen diese dann beim Laden auf (hier ist die Größe der einzelnen Frames durch frameWidth und frameHeight angegeben).

```
1  function preload() {
2      this.load.image('sky','assets/sky.png');
3      this.load.spritesheet('dude','assets/dude.png', { frameWidth: 32,
   frameHeight: 48});
4      this.load.image('platform','assets/platform.png');
5  }
```

5. Create-Methode

Nach dem Preload wird dann die Create-Methode aufgerufen. Diese soll die geladenen Assets einsetzen um so die Spielewelt aufzubauen.

```
1  function create() {
2      scene = this; //Die Variable scene liefert einen Verweis auf this (scene
   = this)
3      initWorld(); //Aufruf von initWorld
4      initPlayer(); //Aufruf von initPlayer
5      //player soll mit den platforms interagieren, d.h. ein Collider wird
   erzeugt
6      this.physics.add.collider(player, platforms);
7       //Variable cursors wird mit den Pfeiltaste initialisiert
8      cursors = this.input.keyboard.createCursorKeys();
9  }
```

5.1. initWorld()

```
function initWorld() {
    //Grafiken haben den Haltepunkt in Phaser in der Mitte.
    //Dies kann mit setOrigin geändert werden.
    scene.add.image(0,0,'sky').setOrigin(0,0); //Sky-Grafik dazufügen an Position 0,0
    platforms = scene.physics.add.staticGroup(); //Statische Gruppe für Plattformen
    //Die Grafik Plattform wird auf 400,400 hingesetzt, mit setScale(2) um den
    //Faktor 2 vergrößert. Mit refreshBody wird auch der Physikalische Body
    //angepasst.
    platforms.create(400,400,'platform').setScale(2).refreshBody();
    //Eine weitere Plattform
    platforms.create(600,300,"platform");
}
```

5.2. initPlayer()

Der Player wird als sogenanntes Sprite erzeugt und hinzugefügt. Mit `setCollideWorldBounds(true)` wird festgelegt, dass er bei den Weltgrenzen nicht mehr weiter gehen kann.

```
player = scene.physics.add.sprite(100,100,'dude');
player.setCollideWorldBounds(true);
```

Für den Player werden nun in der Szene Animationen angelegt. Diese brauchen einen key, damit man sie aufrufen kann, dann eine Reihe von Frames, die für die Animation hergenommen werden (hier z.B. 0 bis 3), eine Framerate, welche die Geschwindigkeit der Animation festlegt und eine Wiederholunszahl (-1 bedeutet ständig wiederholen)

```
scene.anims.create({
        key: 'runleft',
        frames: scene.anims.
            generateFrameNumbers('dude', {start: 0, end: 3}),
        frameRate: 10,
        repeat: -1
    });
```

Eine einfache "Animation" mit nur einem Frame wird folgendermaßen erzeugt:

```
scene.anims.create({
        key: 'stand',
        frames: [{ key: 'dude', frame: 4}],
        frameRate: 10,
        repeat: -1
    });
```

Mit `player.anims.play` und dem Namen des keys in Klammer können nun die einzelnen Animationen angesteuert/gestartet werden.

```
player.anims.play('stand');
```

6. Update-Methode

Die Update-Methode ist jene Methode, die dann in einem Spiel immer wieder aufgerufen wird (bei jedem Bildschirmaufbau). Hier wird auf Veränderungen reagiert, die Steuerung vorgenommen, ...

Bei uns werden die jeweiligen Tasten abgefragt und dann entsprechend reagiert. Beim starten einer Animation gibt es - neben dem key der Animation - noch einen zweiten wichtigen Parameter für uns vom Typ Boolean. Damit wird festgelegt, dass die Animation wirklich immer als ganzes abgespielt werden soll, bevor dann eine nächste Animation darauf angewendet wird. In unserem Fall würde ein Wegnehmen des true dazu führen, dass immer nur das erste Bild angezeigt wird (da ja z.B. bei betätigen einer Taste dieser Befehl x-mal pro Sekunde hintereinander ausgeführt wird und deswegen immer nur bis zum Start, d.h. bis zum ersten Bild käme)

```
function update() {
    if (cursors.left.isDown) {
        player.setVelocityX(-160); //x-Geschwindigkeit auf -160
        player.anims.play('runleft', true); //Spiele Animation "runleft"
    } else if (cursors.right.isDown) {
        player.setVelocityX(160); //x-Geschwindigkeit auf +160
        player.anims.play('runright', true); //Spiele Animation "runright"
    } else {
        player.setVelocityX(0); //x-Geschwindigkeit wieder auf 0
        player.anims.play('stand');  //Spiele Animation "stand"
    }
    if (cursors.up.isDown) {
        player.setVelocityY(-330); //Y-Geschwindigkeit auf -330 (nach oben
hüpfen)
    }
}
```

7. Zusätzliche Einstellungsmöglichkeiten / Features

Hier ein paar zusätzliche Code-Zeilen bzw. Einstellungsmöglichkeiten. Probiere die folgenden Dinge auch selber aus:

Im create ganz zum Schluss könnten wir auch noch die grundsätzliche "Kameraführung" ändern. Ja, richtig, in Phaser gibt es - wie in vielen anderen Spiele-Entwicklungs-Umgebungen - die Möglichkeit unterschiedliche Kameras zu definieren und einzusetzen und dort auch unterschiedliche Effekte zu nutzen (dazu aber später). Eine Kamera ist schon automatisch vorhanden, die man mit `this.cameras.main` ansprechen kann (oder scene.cameras.main). Mit `startFollow` sagen wir dieser, dass sie dem player folgen soll. Setzt man jetzt für den Player auch noch das `setCollideWorldBounds` auf `false` (im `initPlayer`) hat man jetzt aber das Problem, dass der Spieler aus der Welt herausfliegen kann (ins Nirvana)

```
this.cameras.main.startFollow(player);
```

Deswegen könnte man mit folgenden 2 Zeilen die gesamte Weltgröße und - damit es optisch auch passt - den maximalen Wirkungsbereich der Kamera festlegen (setBounds - Grenzen setzen). Hier geht das jeweils vom Punkt 0,0 (x,y) bis zum Punkt 2000,600:

```
this.physics.world.setBounds(0,0,2000,600);
this.cameras.main.setBounds(0,0,2000,600);
```

Ein weiterer netter Effekt, den man auch noch verwenden kann, ist es, dem Spieler wenn er irgendwo aufkommt einen Bounce-Effekt zuzuweisen kann. Dies könnte man z.B. im `initPlayer` mit folgendem Befehl machen:

```
player.setBounce(0.2);
```